Info 206: Computing

Lecture 1

Sorting and Searching

September 3, 2014

Considers Google's problems

- Vast amounts of data (Petabytes)
- Need to process that information very quickly
- This is the <u>scalability</u> problem
- Tools that Google has
  - Fast algorithms (this lecture)
    - Searching, sorting
  - Lots of computers (later in semester)
    - MapReduce

Three key algorithms for big data

- Searching
- Sorting
- MapReduce (will discuss later in semester)

Algorithm

- Algorithm: abstract technique for computation
  - Not tied to a particular programming language

- A computer program implements the algorithm
  - Tied to a particular language

Scalability

- We need to think big: scalability
- Key concern: running time
- Key concern: parallelization
  - (Will discuss later in semester)

Running time

- Running time depends on problem size
  - Sorting a billion items takes more time than sorting a hundred items
- Want a way to describe running time
- Problem – computers vary tremendously in speed
- Solution: “Big-O” notation

$O(f(n))$ - Intuition

- We have a program that takes input of size n
  - Tricky part – what exactly does n measure?
- If n is big enough ($n > N$ for some N)
- There is some constant c such that
- Running time $< cf(n)$ $(n > N)$
- Then running time is $O(f(n))$

$O(f(n))$ – Formal definition

$g(n)$ is $O(f(n))$ if there exists an N and c

such that for all $n > N$

$g(n) < cf(n)$

Example - searching

- Linear search
- Search a list one by one

```
def search (x, nums):
    for i in range(len(nums)):
        if nums[i] == x #item found
            return i    #return index
    return -1           #item not found
```

How many times through loop

```
def search (x, nums):
    for i in range(len(nums)):
        if nums[i] == x #item found
            return i    #return index
    return -1           #item not found
```

- How many times through loop
- `len(nums)`
- $O(n)$

Binary search

```
def search (x, nums):
      low = 0
      high = len(nums) – 1
      while low <= high:              # still searching?
            mid = (low+high)/2        # middle item
            item = nums[mid]
            if x == item:             # found it
                  return mid          # return index
            elif x < item:            # x in lower 1/2
                  high = mid – 1      # adjust high
            else:                     # x in upper 1/2
                  low = mid + 1       # adjust low
      return -1                       # x is not there
```

Running time of binary search

- Each round reduces search size by ½
- Like removing a bit from a binary number
- # of times through loop = # of bits in `len(nums)`written as a binary number
- We call this $\log n$ (logarithm of n)

Some basic O running times

notation	name
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

- Don't confuse polynomial $O(n^c)$ with exponential $O(c^n)$

Sorting algorithms

- How do we sort n items?
- Two algorithms:
  - Bubble sort
  - Quicksort
- See animations at http://bit.ly/info206sort

Bubble sort

- (See animation)

Bubble sort running time

- Round 1: $(n-1)$ possible swaps
- Round 2: $(n-2)$ possible swaps
- …
- Round $(n-1)$: 1 possible swap

- What is $1 + 2 + ... + (n-2) + (n-1)$?
- We use Gauss's trick

Adding numbers

$$1 + 2 + \ldots + (n-2) + (n-1)$$

$n + n$ [$(n-1)/2$ times]

Sum = $n(n-1)/2$

Bubble sort running time

- Round 1: $(n-1)$ possible swaps
- Round 2: $(n-2)$ possible swaps
- …
- Round $(n-1)$: 1 possible swap

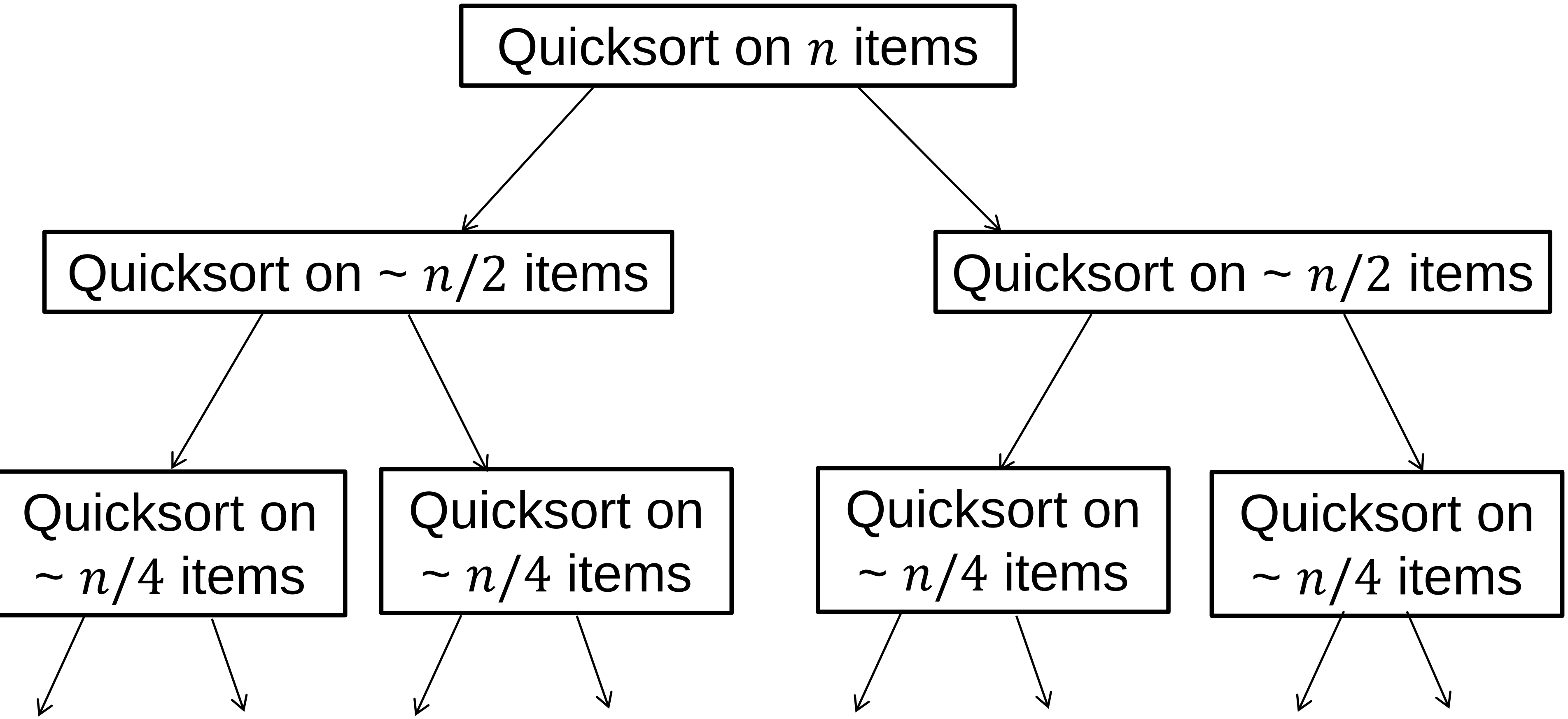
- $1 + 2 + ... + (n-1) = n(n-1)/2$
- $O(n^2)$

Quicksort

- (See animation)

Quicksort idea

- Split sorting problem into two subproblems
- If we are lucky, they subproblems are ½ size
- Recursion



A benefit of recursion

- “Divide and conquer”
- Split a big problem into smaller problems
  - Keep on doing it again and again
- Those smaller problems can be assigned to different computers
- Parallelization!

Running time of quicksort (lucky case)

- Depends on how lucky we are
- Lucky case:
- Round 1: $\sim n$ items compared w/ pivot (n)
- Round 2: $2 \times \sim n/2$ items compared w/pivot (n)
- Round 3: $4 \times \sim n/4$ items compared w/pivot (n)
- …

- So $n \times$ (# of bits in n) = $n \log n$ comparisons
- $O(n \log n)$

Running time of quicksort (worst case)

- Worst case
- Round 1: $(n-1)$ items compared w/pivot
- Round 2: $(n-2)$ items compared w/pivot
- Round 3: $(n-3)$ items compared w/pivot
- …
- This time we need $n-1$ rounds!
- Running time =
- $1 + 2 + ... + (n-1) = n(n-1)/2$
- $O(n^2)$

Usually we are lucky

- When quicksort input random $\rightarrow$mostly lucky
- “Average” running time quicksort is $O(n \log n)$
- This is also <u>the optimal sorting algorithm</u>

- Unlucky case: sorted input (or reverse sorted or partially sorted)
- We can address by choosing pivot randomly

Next lecture

- Arrays and lists